Effective Sampling-Driven Performance Tools for GPU-Accelerated Supercomputers

Milind Chabbi, Karthik Murthy, Michael Fagan, and John Mellor-Crummey
Department of Computer Science, Rice University Houston, TX, USA
{milind.chabbi, karthik.murthy, mfagan, johnmc}@rice.edu

ABSTRACT

Performance analysis of GPU-accelerated systems requires a system-wide view that considers both CPU and GPU components. In this paper, we describe how to extend system-wide, sampling-based performance analysis methods to GPU-accelerated systems. Since current GPUs do not support sampling, our implementation required careful coordination of instrumentation-based performance data collection on GPUs with sampling-based methods employed on CPUs. In addition, we also introduce a novel technique for analyzing systemic idleness in CPU/GPU systems. We demonstrate the effectiveness of our techniques with application case studies on Titan and Keeneland. Some of the highlights of our case studies are: 1) we improved performance for LULESH 1.0 by 30%, 2) we identified a hardware performance problem on Keeneland, 3) we identified a scaling problem in LAMMPS derived from CUDA initialization, and 4) we identified a performance problem that is caused by GPU synchronization operations that suffer delays due to blocking system calls.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Measurement techniques, Performance attributes.

General Terms

Performance, Measurement.

Keywords

Heterogeneous architectures, Performance analysis, Call path profiling, CPU-GPU blame shifting.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

SC '13, November 17 - 21 2013, Denver, CO, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2378-9/13/11$15.00.
http://dx.doi.org/10.1145/2503210.2503299

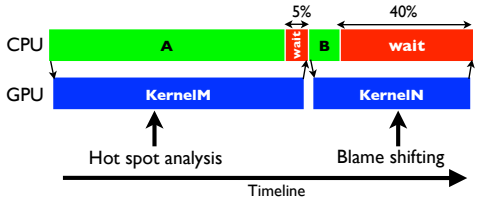

Figure 1: Here we consider a timeline for a CPU-GPU execution. The CPU executes code fragments A and B, and spends a total of 45% of its time waiting. Tuning GPU KernelN has the potential for a larger reduction in execution time than tuning GPU KernelM. Hot spot analysis would identify KernelM, the longer of the two, as the most promising candidate for tuning. Our CPU-GPU blame shifting approach would highlight KernelN since it has a greater potential for reducing CPU idleness.

1. INTRODUCTION

Emerging supercomputers are increasingly employing GPU accelerators. Not only do these GPU-accelerated systems deliver higher performance than their counterparts built with conventional multicore processors alone, but these accelerated systems also deliver improved power efficiency. The increasing use of such GPU-accelerated systems has motivated researchers to develop new techniques to analyze the performance of these systems.

To develop useful performance tools for GPU-accelerated systems, we needed to answer two questions: 1) What data do we want to collect? and 2) How do we want to collect it?

To date, much of the recent work on performance analysis of heterogeneous architectures, e.g., [18, 21], has focused on identifying performance problems in GPU kernels. While identifying GPU kernel-level issues is important, this is only one aspect of the larger problem. Whole application performance analysis is equally important for tuning large GPU-accelerated applications. Such analysis requires a system-level view of performance data. Hence, the data collection question reduces to deciding what kinds of *system-level* analyses can best augment standard component-level profiles and traces.

Studies by Luk et al. [10] and Song et al. [22] have demonstrated that dynamically partitioning an application's work between CPU and GPU is important for delivering high efficiency for a variety of applications. Consequently, we devel-

oped an analysis technique to address the work-partitioning issue. Evaluating the effectiveness of an application's work partitioning is a systemic question. It is not easily addressed by focusing on individual components.

Any tool that focuses on *hot spot analysis* can only quantify *where* a program spends its resources. Each component may have different hot spots. At best, hot spot analysis measures and reports the symptoms of performance problems. Hot spot analysis doesn't necessarily guide the developer towards root causes of performance problems in GPU-accelerated applications. The sample shown in Figure 1 highlights this point. If the CPU code executing during the interval labeled `A` cannot be tuned further, then improving `KernelM`, a GPU kernel whose execution is overlapped with `A`, will not shorten the execution by more than 5%—the time that the CPU sits idle awaiting the results of `KernelM`. However, in the same application, the CPU sits idle for 40% of the execution awaiting the completion of GPU `KernelN`; hence, tuning `KernelN` could reduce the execution time by up to 40%. Hot spot analysis would point to `KernelM` as the most time consuming GPU kernel, and thus fail to guide a programmer to `KernelN`. `KernelN` represents a better opportunity for tuning. This problem is exacerbated in full applications with several kernels and more complicated execution schedules.

To address the limitations of hot spot analysis, we supplement it with novel systemic *idleness* analysis. Our idleness analysis identifies *CPU* code regions that cause *GPU* resources to sit idle. Symmetrically, our approach also pinpoints *GPU* kernels that cause *CPU* cores to sit idle. Moreover, our analysis quantifies the amount of idleness due to each offending CPU code region or GPU kernel. Normally, this sort of systemic analysis would require postmortem analysis of execution *traces*. Our analysis, however, requires only a *profile*. The reason the idleness analysis can be done without traces is due to a technique that we developed called *CPU-GPU blame shifting*.

For the example shown in Figure 1, our blame-shifting strategy identifies GPU `KernelN` as a promising target for tuning. Furthermore, our strategy *quantifies* that tuning `KernelN` could improve performance by no more than 40%.

Another systemic problem that we discovered during the course of this work concerns the effect of blocking system calls on application performance. Much to our surprise, this phenomenon seems to have an especially deleterious effect on CPU calls to GPU interface routines. To correctly identify and quantify this kind of performance problem required us to determine when a given stall was due to blocking system calls. Unlike the idleness analysis previously described, our stall analysis requires traces.

CPU-GPU blame-shifting and stall analysis complement each other and enable idleness analysis from different perspectives. For large-scale applications, while CPU-GPU blame-shifting identifies idleness within a node* (*intra-node* idleness analysis), the stall analysis identifies idleness across nodes (*inter-node* idleness analysis).

Ideally, we would prefer a low-overhead sampling-based approach for data collection. However, GPUs do not yet support sampling-based measurement methodology, so we had to use an instrumentation approach to gather GPU performance data. What we developed is a *sampling-driven* approach. In our sampling-driven approach, CPU performance data is collected using asynchronous sampling, but the sampling engine takes on additional responsibility of inspecting GPU state information maintained with instrumentation. We developed this more complex methodology primarily to enable the blame-shifting idleness analysis described previously.

We note that both our analysis techniques and our sampling-driven measurement methodology are fully general. For the work described in this paper, however, we implemented our ideas to support analysis of CUDA programs.

The remainder of this paper describes our novel analysis techniques and the sampling-driven implementation in more detail. The paper is organized as follows: Section 2 gives general background plus a summary of specific work that forms the basis for our tool. Section 3 describes our general approach, and further details our blame-shifting and stall analysis. Section 4 sketches the details of our implementation. Section 5 describes case studies that illustrate the effectiveness of our techniques. Section 6 gives a brief overview of the methods and capabilities of other performance tools. Finally, Section 7 summarizes our conclusions and discusses possibilities for future work.

2. BACKGROUND

In this section, we describe diverse areas that constitute the background needed for understanding this work. First, we introduce our terminology. Next, we present challenges for performance measurement tools on heterogeneous systems. Then, we describe both NVIDIA's CUDA programming model, and NVIDIA's CUPTI performance tools interface. Finally, we describe the open source HPCTOOLKIT performance tools, which form the basis of our work.

2.1 Terminology

We use the term *heterogeneous* to mean architectures that include both CPU and GPU compute resources. We use the term *hybrid* to describe a code that executes in part on CPU(s) and in part on GPU(s). A *task* is an asynchronous GPU activity, such as a kernel or a data transfer. A task is *outstanding* when it has been enqueued for execution, but not complete. A task is *active* if it has begun execution, but not complete. *Profiling* aggregates performance metrics over time and attributes them to code regions. *Tracing* records execution events in temporal order. For convenience, we refer to our GPU extensions to HPCTOOLKIT as G-HPCTOOLKIT.

2.2 Measurement Challenges on Heterogeneous Platforms

Heterogeneous systems today present several challenges for performance tools:

- **Asynchrony:** GPU activities run asynchronously with respect to the CPU and deliver no notification when they start or end.
- **Resource sharing:** A GPU can execute tasks from multiple threads of a process, as well as multiple MPI processes of an application simultaneously.
- **Minimal infrastructure:** Today's GPUs lack hardware support for performance monitoring using asynchronous sampling. Also, current APIs for GPU per-

* Section 4.5 details multiple processes sharing a node.

formance measurement are missing several key capabilities needed to support whole program analysis.

2.3 CUDA and CUPTI Overview

NVIDIA's CUDA programming model [14] enables programmers to express data parallel computations and map them onto GPUs as tasks. On a heterogeneous architecture, after launching an asynchronous task, a CPU thread can continue execution or immediately wait for completion of the GPU task. CUDA applications manage concurrency through *streams*. A stream is a sequence of tasks that execute in order on a GPU. Tasks from different streams may interleave arbitrarily or run concurrently. CUDA *events* are lightweight markers in a CUDA stream. Events can be queried to inquire about their completion. An event in a stream may not complete until all tasks preceding it in the same stream complete.

NVIDIA's CUPTI performance tools interface [13] is a framework that supports performance analysis of CUDA codes. CUPTI provides the ability to collect GPU hardware counter values. Unlike CPU hardware counters, GPU hardware counters do not deliver an interrupt when they overflow. CUPTI supports attributing hardware counter measurements to GPU activities by providing callbacks upon entry/exit of any CUDA API routine. CUPTI's *Activity API* can trace execution of tasks. For each task that executes, it logs a record that contains the task start and end times. Activity records can be examined only at synchronization points.

2.4 HPCToolkit

HPCToolkit [1] is an open source suite of tools that supports measurement, analysis, attribution, and presentation of application performance for parallel programs. HPCToolkit has three features that characterize its efficacy for CPU programs. First, HPCToolkit collects almost all performance data using asynchronous sampling of timer and hardware counters, employing instrumentation only when absolutely necessary. HPCToolkit's overhead for performance data collection is typically less than 5% [24]. Second, HPCToolkit attributes performance metrics to full calling contexts for CPU code. In addition, HPCToolkit maintains these calling contexts in a compact Calling Context Tree (CCT) representation [2], leading to compact profiles, and exceedingly small execution traces [23]. Third, by using on-the-fly binary analysis, HPCToolkit works on multilingual, fully-optimized, statically or dynamically linked applications and supports a wide variety of programming models. This on-the-fly binary analysis enables HPCToolkit to attribute costs to both an application, and its constituent runtime libraries without instrumenting either of them.

Prior to this work, HPCToolkit provided no support for analysis of hybrid codes. In this paper, we describe how we address that shortcoming via G-HPCToolkit.

3. NEW ANALYSIS TECHNIQUES

As we alluded to in the introduction, we augment HPCToolkit's standard profiles and traces with two new analyses to pinpoint bottlenecks in execution of hybrid programs: idleness analysis and stall analysis. Section 3.1 covers the idleness analysis. Section 3.2 covers the OS blocking stall analysis.

3.1 Idleness Analysis via Blame-shifting

In hybrid codes, the execution schedule for CPU and GPU tasks governs tuning opportunities and helps set performance expectations. When code regions are well overlapped, tuning just one part provides little overall performance improvement if the other part is already well tuned. For example, tuning a lengthy GPU kernel is unnecessary if the CPU only requires its results long after completion of that kernel. Tuning code regions whose executions don't overlap, however, can reduce the critical path and thereby improve the overall running time. In hybrid codes, when a GPU is idle (symptom), CPU threads (cause) are responsible for not keeping it busy with sufficient number of asynchronous tasks. Similarly, when a CPU thread is waiting (symptom) for GPU tasks to produce results, the active GPU tasks (cause) are responsible for blocking the CPU thread from making progress.

Our approach for identifying causes of idleness in heterogeneous applications was inspired by the *blame shifting* technique pioneered by Tallent et al. [25]. Tallent et al. were concerned with lock contention. Their key insight was to attribute blame to lock *holders* for the idleness of threads waiting for lock acquisition. We apply this idea of shifting blame for idleness in one part of the system to concurrent computation elsewhere in the system to analyze how applications use the compute resources of heterogenous architectures. The result is a strategy we call *CPU-GPU blame shifting*. In *CPU-GPU blame shifting*, we blame code executing on the non-idle resource (e.g., a kernel executing on a GPU) when a symptom of idleness is detected (e.g., CPU waiting at `cudaDeviceSynchronize`). We *quantify* the causes of idleness as follows:

- CPU code regions that execute when a GPU is idle accumulate blame proportional to the time that the GPU was idle during their execution.
- GPU tasks that execute when a CPU thread is awaiting their completion accumulate blame proportional to the time that the CPU thread waited for their completion.

In practice, "blame" is represented by the wall clock time or cycles for which a resource is idle. Code region(s) that are frequently responsible for causing idleness accrue more blame than other regions and thereby warrant the programmer's attention. We characterize hybrid codes with the metrics presented in Table 1. `CPU_IDLE_CAUSE (CIC)` and `GPU_IDLE_CAUSE (GIC)` are the blame shifting (cause) metrics. `CPU_IDLE (CI)` is an idleness symptom metric. `GPU_EXECUTION_TIME (GET)`, `H_TO_D_BYTES (H2D)`, and `D_TO_H_BYTES (D2H)` are useful hot spot metrics. Each metric adds up to provide the total time spent in that state. We attribute each metric to a full call path, hence the programmer can easily identify inclusive and exclusive costs at each level in the call path. A key attraction of this technique is that the analysis and attribution can be performed on-the-fly without having to record full execution traces.

In this work, we focus only on CPU-GPU idleness. We do not consider the cases when a CPU-thread is idle for a different reason; such as waiting for another CPU-thread or I/O activity. Also, this paper does not explore gathering GPU hardware performance counter measurements of individual kernels. HPCToolkit, like other tools in Section 6, uses

Metric	Qualification	Context of attribution	Quantification (amount)
CPU_IDLE (CI)	CPU waiting for GPU	CPU code region e.g. cudaDeviceSynchronize	Time spent waiting for GPU
CPU_IDLE_CAUSE (CIC)	CPU waiting for GPU	Launch contexts of active GPU tasks (blame)	Time the task was active when CPU was waiting
GPU_IDLE_CAUSE (GIC)	GPU idle	CPU code regions (blame)	Time when GPU was idle
GPU_EXECUTION_TIME (GET)	Length of GPU task	Launch context of the GPU task	Time taken for the GPU task
H_TO_D_BYTES (H2D)	Host to device data xfer	Launch context of the CPU-to-GPU data transfer	bytes transferred
D_TO_H_BYTES (D2H)	Device to host data xfer	Launch context of the GPU-to-CPU data transfer	bytes transferred

Table 1: Metrics for performance analysis of hybrid codes. *IDLE_CAUSE metrics are computed using blame shifting.

the CUPTI interface to gather GPU performance counter values for individual kernels.

3.2 Stall Analysis

Our stall analysis technique detects when performance is degraded by undue delays in blocking system calls. Even if excessive blocking occurs infrequently in each process, it can dramatically degrade the performance in parallel systems [19]. For example, suppose one process is delayed due to a blocking system call. If the delay occurs before arrival at a barrier, then the entire system must wait for the late arriver. We emphasize that we are seeking *sparse* blocking phenomena. If *most* processes are delayed before arriving at a barrier, this set of blocking events is not a candidate for stall analysis. Non-sparse blocking phenomena are easily captured as part of blame shifting. Moreover, the sparse stall phenomena may repeat many times while an application executes. Our stall analysis calculates the cumulative effect of such stalls. Unlike blame shifting, stall analysis is performed via postmortem inspection of execution traces.

Detecting a sparse stall event consists of three parts: 1) detect that a stall has occurred in at least one process, 2) confirm that the set of stalls for the given time interval is sparse, and 3) confirm that the sparse set affects the performance of the full application. To detect that an undue delay in a blocking system call has occurred in a given trace, we rely on a regularity property of time-based periodic sampling: samples should occur at regular intervals. While some variation in the length of sampling intervals is normal, any "significant" lengthening of the time intervals between samples indicates that the periodic timer interrupts are not being reported by the OS, which means that blocking has occurred in that interval.

To determine when a given interval in a trace is unusually long, we find the median interval span in the given trace. We mark a span as unduly long if its length is more than 10 times the median.

To determine if a given unduly long interval is part of a sparse set, we count the total number of unusually long intervals in each trace that occur within the same time period. If less than 10% of the traces have unusually long intervals for the time period in question, then this set of unusually long intervals is a sparse set.

Finally, to determine if the sparse set affects overall system performance, we check the calling context of each sample in the set of traces that are *not* in the sparse set. If each sample has a barrier or collective operation in the calling context, then the sparse set meets all the criteria to be deemed an unusually long stall. We freely acknowledge that the 10x median rule for "unusually long" and the less than 10% rule for classifying a set as sparse is ad hoc. These should be tunable parameters. In our prototype, however, we restricted ourselves to the aforementioned thresholds.

To quantitatively assess the cumulative impact of all unusually long stalls, we merge delays from two or more processes that have overlapping stalls and consider it as one larger stall starting at the earliest start time and ending at the latest end time of the overlapping stalls. It is easy to accomplish this by using interval trees to compute the union of the overlapping time ranges. The aggregate value of non-overlapping time ranges accumulated in the interval tree provides the total system-affecting blocking time and its ratio with respect to the total execution time of the program provides the percentage of time that the execution was impacted due to blocking system calls.

4. IMPLEMENTATION

We organize the implementation details of our idleness analysis as follows: Section 4.1 describes a basic mechanism for our implementation of CPU-GPU blame shifting. Section 4.2 details a strategy to accommodate blame shifting inside a CUDA call. Section 4.3–4.5 gradually develop CPU-GPU blame shifting for multi-stream, multi-thread, and multi-process shared GPU configurations. Finally, section 4.6 describes support for GPU tracing in HPCToolkit. In rest of this paper, we use the terms "CPU-GPU blame shifting" and "blame shifting" interchangeably.

4.1 Basic CPU-GPU Blame Shifting

G-HPCToolkit employs sampling-based performance measurement of code running on each CPU core in combination with lightweight instrumentation of GPU operations. Figure 2 provides a schematic view of G-HPCToolkit. To monitor asynchronous GPU tasks, we use a two-part technique. First, we insert CUDA *events* before ("start") and after ("end") all GPU task launches. This technique is similar to the "Event Queue Method" independently developed by Malony et al. [11]. Inserting these events requires instrumenting CUDA task-launch functions. While CUPTI provides suitable instrumentation hooks, we manually wrapped CUDA's kernel launch API using library interposition [12]. In addition, we wrap functions that manage streams, allocate memory, and perform copies or synchronization. We chose this method to circumvent host-thread serialization that occurs when using pre-5.0 versions of CUPTI.

The second part of our two-part technique involves querying the status of these events. Querying takes place periodically, when a CPU sample occurs. At any time, we infer that a task is active if its "start" event is complete, but its "end"

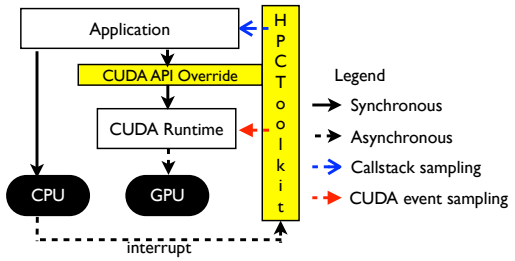

Figure 2: Schematic view of G-HPCToolkit.

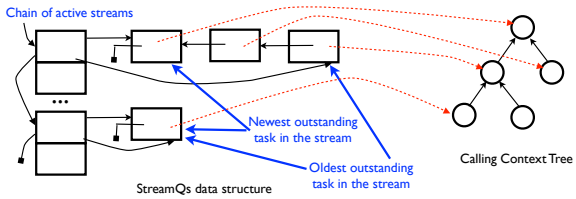
Chain of active streams
Newest outstanding task in the stream
Oldest outstanding task in the stream
StreamQs data structure
Calling Context Tree

Figure 3: Data structure used to maintain the association between GPU tasks and CPU contexts.

event is incomplete. A task is complete if its "end" event is complete. Querying can happen either during sampling or inside one of the CUDA-functions we wrap. By measuring the elapsed time between recorded timestamps for the completion of "start" and "end" events, we get a close estimate of the execution time of a task. It is worth mentioning that events, in addition to a complete/incomplete boolean flag, carry a completion timestamp which can be queried any time after the completion of the event. To confirm the accuracy of our technique, we compared the deviation of kernel timings as measured by CUDA events vs. CUPTI's Activity API for 26 unique kernels in the LULESH 1.0 [8] benchmark. At the sample rate of one per millisecond, the geometric mean of percentage deviations from the data reported by ActivityAPI over all LULESH kernels was ~1%, which we deem acceptable.

We store the "start" and "end" events of each task, along with a pointer to the call path that launched the task in an auxiliary data structure that we call `StreamQs`. G-HPCToolkit's `StreamQs` data structure, shown in Figure 3, is an array of queues, where each queue represents the tasks issued on a GPU stream. The head and tail of each queue in `StreamQs` represent the oldest and newest outstanding tasks issued on that stream respectively. A new GPU task can be enqueued at the tail of a queue in a constant time. On each sampling event, the CPU queries the completion status of the "end" event of the task at the head of each active stream, which can be accessed in constant time. If the "end" event is complete, the query moves forward through the list until it reaches an incomplete event in that stream. `StreamQs` may be accessed and updated *synchronously* or *asynchronously* by all threads in a process. *Synchronous* accesses happen during the execution of instrumentation inserted in wrapped functions, whereas *asynchronous* accesses happen during sampling interrupts.

Figure 4 sketches our approach for sampling-based performance analysis of GPUs for a single CPU-thread, single

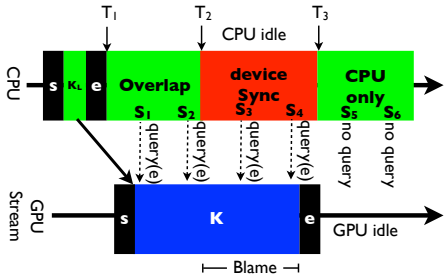

Figure 4: Proxy-based GPU sampling.

GPU-stream case. We use the `StreamQs` data structure to help transfer blame to the appropriate CPU and GPU contexts. Let the calling context at a point P in CPU execution be represented as $c(P)$. Let the calling context for the CPU-side launch point of a GPU task K be represented as $c(K_L)$. At $c(K_L)$, in Figure 4, G-HPCToolkit inserts the start event s and end event e surrounding the task K through the wrapped `cudaLaunch` API. A new node $< s, c(K_L), e >$ is enqueued into a queue representing the appropriate stream in the `StreamQs` data structure.

At time T_1, the CPU starts to execute the code marked as *overlap* concurrently with kernel K on a GPU stream. On interrupts S_1 and S_2 during this interval, G-HPCToolkit queries the status of the end event e for the oldest outstanding task K, and in this case, infers that K is active. Therefore, no blaming occurs during this stage.

At time T_2, CPU calls `cudaDeviceSynchronize` to wait for the completion of tasks. G-HPCToolkit's wrapped version of the function sets a thread-local flag (`isAtSync`) indicating that the CPU thread is idle. The blame attribution in this region follows the deferred blame shifting strategy discussed in Section 4.2, which leads to blaming kernel K with the amount $(T_3 - T_2)$ as the cause of CPU wait time. This is accomplished by incrementing the `CPU_IDLE_CAUSE` metric for $c(K_L)$ by $(T_3 - T_2)$.

At T_3, the return from `cudaDeviceSynchronize` causes G-HPCToolkit to query e and infer the completion of task K, thereby dequeueing it from `StreamQs`. Also, we unset the `isAtSync` flag indicating that the CPU is active. Sampling at S_5 declares the GPU idle due to no outstanding GPU-tasks. Samples taken at S_5 and S_6 blame their respective CPU contexts for keeping the GPU idle, which is accomplished by incrementing their `GPU_IDLE_CAUSE` metric.

4.2 Deferred Blame Shifting

To sample GPU events, G-HPCToolkit needs to call `cudaEventQuery` from within a signal handler. When an asynchronous sample occurs, if the CPU thread is inside a CUDA API function such as `cudaDeviceSynchronize` that already holds a CUDA runtime lock, then calling `cudaEventQuery` will cause it to attempt to acquire the same lock leading to a deadlock.

To avoid deadlock in this circumstance, we devised a technique that we call *deferred blame shifting*. In deferred blame shifting, when an asynchronous sample occurs, if the thread is already inside a CUDA call, which can be inferred by inspecting `isAtSync` flag, G-HPCToolkit's timer interrupt handlers do not query for the completion of outstanding

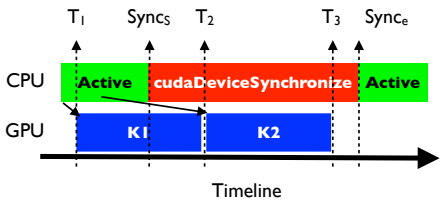

Figure 5: Deferred blame attribution: A technique to attribute blame to GPU kernels when CPU is inside a blind spot such as `cudaDeviceSynchronize`.

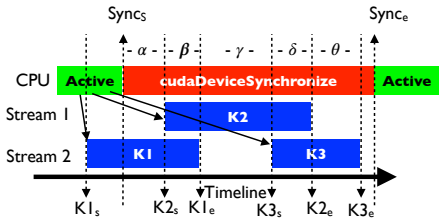

Figure 6: Blame attribution with multiple streams.

tasks and remain in a blind spot with respect to GPU activities. In the example in Figure 4, the time interval between T_2 to T_3 is a blind spot. On returning from a CUDA API call, the wrapped function examines outstanding GPU tasks in each active stream of the `StreamQs` data structure looking for ones that are complete and proportionately blames them for causing the CPU idleness. The ability of events to record completion timestamps for a later query comes in handy here.

To illustrate the deferred blame shifting mechanism, consider Figure 5. From time $Sync_s$ to $Sync_e$, our tool cannot inquire about GPU activity. Just before returning from the wrapper for `cudaDeviceSynchronize`, however, we query for the completion of $K1$ and $K2$. We increment the `CPU_IDLE` metric by $(Sync_e - Sync_s)$, attributing the idleness to the CPU context that called `cudaDeviceSynchronize`. Having obtained the start and end times of $K1$ and $K2$, we now *blame* CPU idleness (by incrementing `CPU_IDLE_CAUSE` metric) at $c(K1_L)$ for the amount $(T_2 - Sync_s)$ and $c(K2_L)$ for the amount $(T_3 - T_2)$. The blame for the $(Sync_e - T_3)$ interval is still unattributed – both CPU and GPU are idle here. We note that this part of the idleness is the delay between the end of GPU activity and the return from the CUDA synchronization call waiting on it. In our blame-shifting methodology, we consider this delay to be primarily GPU idleness, and we blame it on the CPU context calling `cudaDeviceSynchronize`. A later case study in Section 5.1 with LULESH demonstrates a scenario where such a situation arises with `cudaFree`.

4.3 Blame Shifting with Multiple Streams

When concurrent kernels are active on multiple streams, CPU idleness is apportioned across all active kernels. Consider the situation shown in Figure 6, where the CPU thread is idle from time $Sync_s$ to $Sync_e$, and three kernels `K1`, `K2`, and `K3` overlap with this region. Here, the CPU launches `K1` on stream 2, `K2` on stream 1, and `K3` on stream 2, in that order before going into a wait via `cudaDeviceSynchronize`. Blame attribution for each kernel is calculated as follows:

- Blame on `K1` $= (\alpha) + (\frac{\beta}{2})$. α represents the part of `K1` solely responsible for keeping the CPU idle; β represents part of `K1` overlapped with `K2`, and hence the blame is shared by both kernels.
- Blame on `K2` $= (\frac{\beta}{2}) + (\gamma) + (\frac{\delta}{2})$. The first term represents the part of `K1` overlapped with `K2`, and hence the blame is shared. The second term represents the part of `K2` solely responsible for keeping the CPU idle. The third term represents the part of `K2` overlapped with `K3`, and hence the blame is shared.
- Blame on `K3` $= (\frac{\delta}{2}) + (\theta)$. The first term represents the part of `K3` overlapped with `K2`, and hence the blame is shared. The second term represents the part of `K3` solely responsible for keeping the CPU idle.

As before, the delay in return from synchronization $(Sync_e - K3_e)$ is regarded as GPU idleness and blamed on the CPU.

The technique can be implemented in $O(n \log n)$ time complexity by sorting the start and end times of all tasks overlapped in a blind spot, where n is the number of active tasks in the blind spot.

Different CUDA synchronization APIs have different properties. For example, `cudaDeviceSynchronize` waits for all streams to finish; when this API is invoked, we blame CPU idleness on kernels on all streams. On the other hand, `cudaStreamSynchronize` waits for a particular stream to finish, in which case we blame tasks on that stream only.

4.4 Blame Shifting with Multiple Threads

In multi-threaded processes, two cases need to be handled. First, each thread independently blames its CPU context when it observes GPU inactivity. Second, if multiple threads are waiting for one or more GPU kernels to finish, each thread independently blames each kernel that caused its idleness by the amount proportional to that thread's wait time. Multi-thread and multi-stream are composable. The ability to share `StreamQs` helps us achieve blame shifting easily in multi-threaded applications.

4.5 Blame Attribution for Shared GPUs

Software on supercomputers supports running multiple MPI ranks per node sharing a single GPU. Processes do not have a global view of GPU utilization when the same GPU device is shared by multiple MPI ranks, due to different address spaces. In such scenarios, a GPU is idle if and only if none of the processes sharing the GPU have any outstanding kernels scheduled on the GPU. To correctly identify GPU idleness, we introduce a shared counter (`LCounter`), allocated in a shared memory segment created using Linux `shmem` capability. Each time a task is issued to a GPU, that GPU's associated `LCounter` is atomically incremented; the counter is atomically decremented when the task finishes. If the `LCounter` is non-zero, then the GPU is busy on behalf of some process, and G-HPCToolkit will not blame the CPU code on any of the processes. Similarly, to blame CPU idleness on GPU tasks issued by other processes, we employ another shared counter (`SCounter`). `SCounter` is atomically incremented/decremented on entry/exit to/from an idle region (e.g., `cudaDeviceSynchronize`). During execution of a kernel K, the launching process observes `SCounter` and instantaneously blames K.

A known limitation of using events for time measurement is the following: consider two kernels, `K1` and `K2`, along with their start and end events `<s1, K1, e1>` and `<s2, K2, e2>` on two different streams. Let the kernels execute serially, and let the interleaving order be `s1`, `s2`, `K1`, `e1`, `K2`, and `e2`. In such cases, G-HPCTOOLKIT can be fooled into assuming that `K1` and `K2` executed concurrently.

4.6 Lightweight Traces for Hybrid Programs

Tracing involves recording events with timestamps to analyze how an execution unfolds over time. G-HPCTOOLKIT traces CPU-side execution by leveraging the existing infrastructure in HPCTOOLKIT [23]. To monitor CPU activity, HPCToolkit logs a trace record each time a thread receives an asynchronous sample. HPCToolkit's traces consist of a sequence of records, where each record contains a timestamp and the index of a node in a CCT. The path from that node to the root of the CCT represents a full calling context. Each call path is stored only once in the CCT, and as a consequence, the size of HPCToolkit's trace for a thread is proportional to the number of samples it receives during execution.

In contrast to the sampling-based traces HPCToolkit logs for CPU activity, we trace GPU activities by logging records (time/calling context pairs) at the beginning and end of GPU tasks on each CUDA stream. At present, all tasks on a stream are logged into a trace buffer when we notice their completion. Using this approach, the volume of GPU traces for an application is proportional to the number of GPU tasks the application executes. When GPU tasks complete at a rate higher than a chosen sampling rate, we could reduce trace sizes by dropping information about some GPU activities and logging tasks only at a rate proportional to the sampling frequency. A technique we envision is to record only the most recently finished activity on each stream in `StreamQs` into our traces, and drop others that might have finished between the last sample and the current sample. This would result in sampled traces analogous to those HPCToolkit records for CPU activity. It is worth noting that blame shifting is agnostic to trace collection and hence it is unaffected if we drop trace records.

Compared to other tracing tools, our traces are both rich and lightweight. They are rich in the sense that each trace record represents a full calling context. Our trace records enable us to analyze and visualize an execution at multiple levels of abstraction, i.e., different call path depths. Our traces are lightweight in the sense that each trace record itself is compact (only 12 bytes). Our approach of using the index of an out-of-band CCT node to represent the calling context for a trace record enables us to avoid tracing procedure entry and exit events to recover calling context. For GPU-accelerated program executions, HPCToolkit's `hpctraceviewer` graphical user interface presents a trace line for each CPU thread and each GPU stream. For an MPI program `hpctraceviewer` presents such a set of trace lines for each MPI rank.

5. EVALUATION

In this section we evaluate G-HPCTOOLKIT for its efficacy and runtime overhead. To test the efficacy of our techniques, we followed a fairly uniform analysis methodology. For each test case, we profiled it using G-HPCToolkit with CPU-GPU blame shifting activated. Also, we collected traces each time. If the insights gained from these techniques pointed to an opportunity for improvement, we implemented it. If no solution was apparent, but the evidence suggested delays related to blocking system calls, then we ran our stall analysis to confirm (and quantify) the effect of these delays.

In Sections 5.1–5.3, we present case studies that demonstrate unique insights that G-HPCTOOLKIT provides. In Section 5.4, we give a *preliminary* (essentially anecdotal) evaluation of the monitoring-time overhead of G-HPCTOOLKIT. We performed most of our experiments on Georgia Tech's Keeneland Initial Delivery System (KIDS) [26], which is a 120-node HP SL-390 cluster with a Qlogic QDR InfiniBand interconnect. Each node has two Intel Xeon X5660 hex-core CPUs, 24GB memory, and three NVIDIA M2090 GPUs. We also performed some experiments on ORNL's Titan—a Cray XK7 supercomputer. Titan has 18,688 compute nodes linked by Cray's Gemini interconnect. Each compute node has a 16-core AMD Interlagos processor and an NVIDIA Tesla K20X GPU.

5.1 Case Study: LULESH

As part of DARPA's UPHC program, Lawrence Livermore National Laboratory developed the Livermore Unstructured Lagrange Explicit Shock Hydro dynamics (LULESH) mini-application [8]. LULESH is an Arbitrary Lagrangian Eulerian code that solves the Sedov blast wave problem for one material in 3D. In this paper, we study the CUDA-accelerated version of LULESH 1.0 with G-HPCTOOLKIT, which provided us with the following insights:

- The CPU is idle 62% of the wall clock time, and 86% of that time is spent inside `cudaFree`.
- The GPU is idle 35% of the wall clock time. `cudaFree` and `cudaMalloc` called from various contexts account for 48% and 47% (a total of 95%) of GPU idleness respectively.
- There is negligible overlap between the CPU and GPU.

Examining the code regions *causing* CPU idleness (not shown) and GPU idleness (shown in Figure 7) using HPCToolkit's `hpcviewer` provides a clue that the developer used `cudaFree` as a synchronization construct analogous to `cudaDeviceSynchronize` in addition to using it to free the allocated device memory. The pattern of repeatedly allocating and freeing device memory was pervasive in the code and was executed several times in each time step. When sorted by the `GPU_IDLE_CAUSE` metric, `hpcviewer` was able to pinpoint call sites that repeatedly invoked `cudaFree` and `cudaMalloc` as the root causes of GPU idleness.[†] NVIDIA engineers confirmed that `cudaFree` has undocumented dual functionality i.e., it performs `cudaDeviceSynchronize` waiting for all tasks on the GPU to finish, followed by freeing the memory[6]. The CPU-side wait during the synchronization point for the completion of kernels is the primary cause of CPU idleness, *however after synchronization, the CPU is busy freeing the device memory during which the GPU is idle.* Similarly, repeated `cudaMalloc`s are also a cause of GPU idleness. Small amounts of GPU idleness scattered across tens of `cudaFree/cudaMalloc` calls in each iteration add up to contribute ∼30% GPU resource idleness. This

[†]In the figure, we show only the top most contributors, but there are other places where the pattern repeats.

Order by CPU_IDLE_CAUSE	Kernel	CPU_IDLE_TIME μ sec	GPU_EXECUTION_TIME in μ sec (% of total GPU time)	Rank if sorted by GPU_EXECUTION_TIME
1	CalcFBHourglassForceForElems	2.31e+06	2.31e+06 (23.9%)	1
2	CalcHourglassControlForElems	8.49e+05	1.10e+06 (11.4%)	4
3	IntegrateStressForElems	6.73e+05	6.76e+05 (7.0%)	5
4	AddNodeForcesFromElems2	4.16e+05	4.16e+05 (4.3%)	6
5	AddNodeForcesFromElems	3.88e+05	3.88e+05 (4.0%)	7
6	CalcKinematicsForElems	3.17e+05	1.74e+06 (18.0%)	2

Table 2: Blame shifting vs. Hot spot analysis for LULESH kernels.

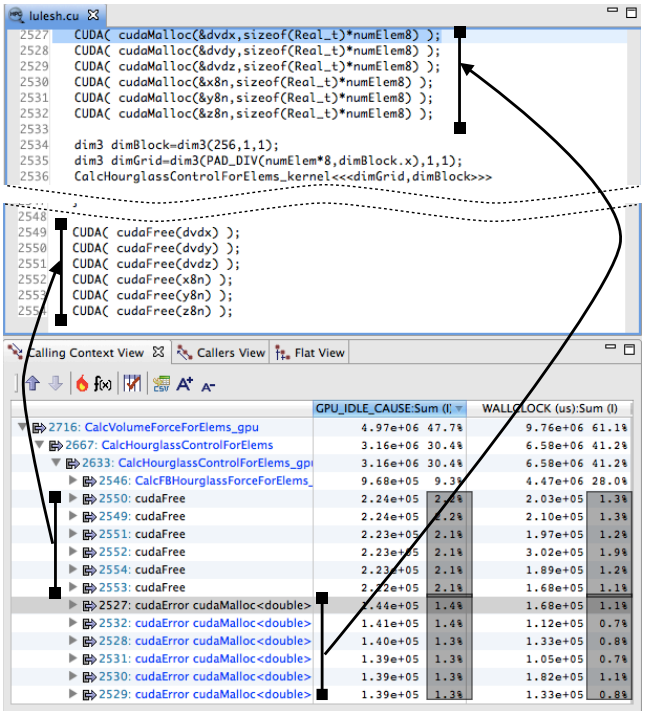
lulesh.cu
2527 CUDA(cudaMalloc(&dvdx,sizeof(Real_t)*numElem8));
2528 CUDA(cudaMalloc(&dvdy,sizeof(Real_t)*numElem8));
2529 CUDA(cudaMalloc(&dvdz,sizeof(Real_t)*numElem8));
2530 CUDA(cudaMalloc(&x8n,sizeof(Real_t)*numElem8));
2531 CUDA(cudaMalloc(&y8n,sizeof(Real_t)*numElem8));
2532 CUDA(cudaMalloc(&z8n,sizeof(Real_t)*numElem8));
2534 dim3 dimBlock=dim3(256,1,1);
2535 dim3 dimGrid=dim3(PAD_DIV(numElem*8,dimBlock.x),1,1);
2536 CalcHourglassControlForElems_kernel<<<dimGrid,dimBlock>>>
2549 CUDA(cudaFree(dvdx));
2550 CUDA(cudaFree(dvdy));
2551 CUDA(cudaFree(dvdz));
2552 CUDA(cudaFree(x8n));
2553 CUDA(cudaFree(y8n));
2554 CUDA(cudaFree(z8n));
Calling Context View
Callers View
Flat View
GPU_IDLE_CAUSE:Sum (I)
WALLCLOCK (us):Sum (I)
2716: CalcVolumeForceForElems_gpu 4.97e+06 47.7% 9.76e+06 61.1%
2667: CalcHourglassControlForElems 3.16e+06 30.4% 6.58e+06 41.2%
2633: CalcHourglassControlForElems_gp 3.16e+06 30.4% 6.58e+06 41.2%
2546: CalcFBHourglassForceForElems_ 9.68e+05 9.3% 4.47e+06 28.0%
2550: cudaFree 2.24e+05 2.1% 2.03e+05 1.3%
2549: cudaFree 2.24e+05 2.1% 2.10e+05 1.3%
2551: cudaFree 2.23e+05 2.1% 1.97e+05 1.2%
2552: cudaFree 2.23e+05 2.1% 3.02e+05 1.9%
2554: cudaFree 2.23e+05 2.1% 1.89e+05 1.2%
2553: cudaFree 2.22e+05 2.1% 1.68e+05 1.1%
2527: cudaError cudaMalloc<double> 1.44e+05 1.4% 1.68e+05 1.1%
2532: cudaError cudaMalloc<double> 1.41e+05 1.4% 1.12e+05 0.7%
2528: cudaError cudaMalloc<double> 1.40e+05 1.3% 1.33e+05 0.8%
2531: cudaError cudaMalloc<double> 1.39e+05 1.3% 1.05e+05 0.7%
2530: cudaError cudaMalloc<double> 1.39e+05 1.3% 1.82e+05 1.1%
2529: cudaError cudaMalloc<double> 1.39e+05 1.3% 1.33e+05 0.8%

Figure 7: Code-centric view of LULESH with `hpcviewer`.

quantifies the expected speedup by eliminating this idleness to be ∼30%.

Figure 8 shows LULESH's pattern for allocating device memory as a function of time. In each time step, at each allocation site, exactly the same amount of device memory is allocated and freed. To eliminate GPU idleness caused by repeated device memory management calls, we hoisted them all outside of the main time step loop. There, we allocated a sufficient number of blocks of sufficient size to meet the peak memory demand. Finally, to replace the synchronization that `cudaFree` provided, we explicitly called `cudaStreamSynchronize`. **This combination of hoisting, pre-allocation, and explicit synchronization reduced the running time for LULESH by 30%.** We reiterate that G-HPCToolkit's `GPU_IDLE_CAUSE` metric identified calls to `cudaFree` *in the application code*. In addition to the running-time benefit, the optimizations improved LULESH's GPU utilization from 65% to 95%. LLNL researchers had independently identified the same performance bottlenecks and admitted a large manual effort in doing so, where as with *CPU-GPU blame shifting* we were able to systematically and automatically identify performance issues in a fraction of time and effort for an unfamiliar code.

Furthermore, Table 2 shows the top six kernels of

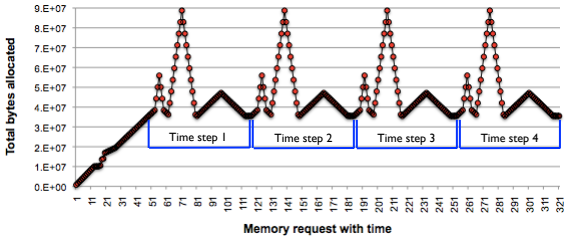

Figure 8: LULESH device-memory allocation graph.

LULESH ordered by their cause for CPU idleness (`CIC` metric). The table also shows, their GPU execution time (`GET` metric) and what would have been their rank order (last column) had they been ordered by `GET` — a hot spot analysis metric. It is worth observing that the the two orderings are quite different. In particular, note that the 2^{nd} rank kernel (`CalcHourglassControlForElems`) in the blame-shift ordering appears as the 4^{th} rank in the hot spot ordering This ranking discrepancy tells us that the `CalcHourglassControlForElems` kernel (ranked 2^{nd} by blame shifting) is a more promising candidate for tuning than the `CalcKinematicsForElems` kernel (ranked 2^{nd} by hot spot analysis).

5.2 Case Study: LAMMPS

LAMMPS [20] is a molecular dynamics code developed by Sandia National Laboratories that simulates biomolecules, polymers, materials, and mesoscale systems. The principal simulation computations in LAMMPS are *neighbor calculation*, *force calculation*, and *time integration*. LAMMPS is parallelized using spatial-decomposition techniques, and employs MPI as the distributed computation infrastructure. LAMMPS-GPU [5] is an accelerated version that offloads neighbor and force computations to GPUs while performing time integration on CPUs.

In Section 5.2.1 we describe how G-HPCToolkit was able to identify a GPU hardware problems on KIDS. Also, in Section 5.2.2 we demonstrate G-HPCToolkit's capabilities for pinpointing scalability losses.

5.2.1 *Pinpointing Hardware Performance Problems*

Figure 9 shows G-HPCToolkit traces from a 64 MPI process execution of LAMMPS-GPU for the Lennard Jones (LJ) benchmark. Each GPU is shared by two processes. Each process has two GPU streams. We filtered the results to show only one GPU stream of each process. A high-level view of the traces shows anomalous behavior on GPU streams associated with two of 64 MPI ranks. Zooming in on these two processes (MPI processes 24 and 25), shows that their host-device data transfers were significantly slower than those of their peers. However, their kernel executions

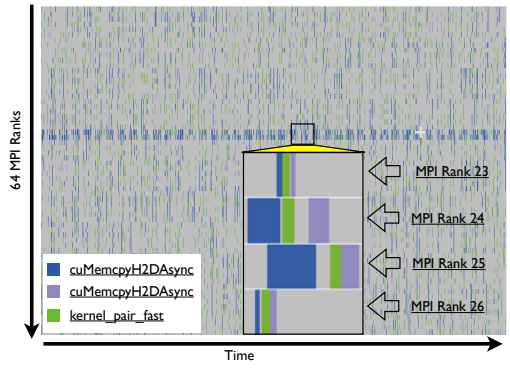

Figure 9: Slow GPUs delaying LAMMPS on KIDS.

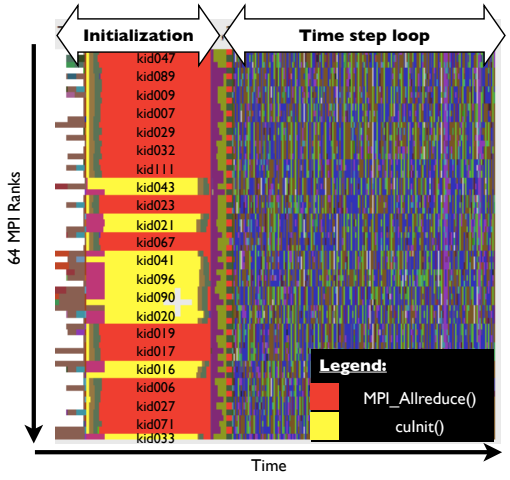

Figure 10: `cuInit` delaying `MPI_Allreduce` in LAMMPS.

were comparable. Metrics for data copies between host and device (`H2D` and `D2H`) for these MPI ranks indicated no difference in the data transfer volume compared to other ranks. This indicated that there might be a hardware problem with the PCI-e bus linking the CPU and GPU on that node hosting those ranks. The KIDS system administrators confirmed that our job was scheduled on node `kid058` that was delivering low PCI-e bandwidth due to an improperly seated GPU. These two slow processes were in turn slowing down the entire execution, since other processes needed to wait for their results at the end of each time step. G-HPCToolkit's lightweight traces in conjunction with the data copy (`H2D` and `D2H`) metrics highlighted the anomalous behavior of the malfunctioning hardware.

5.2.2 Pinpointing Scalability Limiting Factors

Figure 10 represents traces from a 64 MPI process execution of LAMMPS-GPU for the LJ benchmark. In this figure, we filtered all GPU traces and retained only CPU timelines. The left-half of the trace represents initialization and the right-half shows the time step loop. The initialization phase shows that certain processes (e.g. processes on node `kid047`) have a short (~600ms) `cuInit` phase and

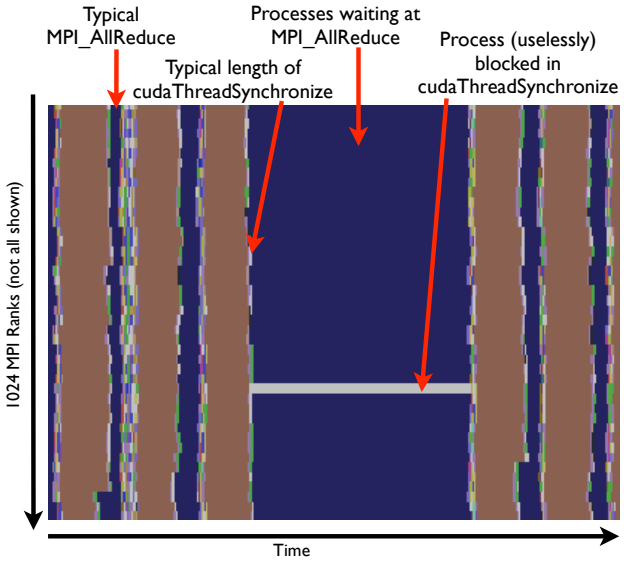

Figure 11: Blocking of CUDA synchronize operations on Titan in LAMMPS.

enter `MPI_Allreduce` waiting for other processes (e.g. processes on node `kid043`) that take much longer (~7s) to finish their `cuInit` phase. Here, slow `cuInit` on some nodes is delaying all processes. When we compared 32 vs. 64 MPI-processes strong scaling LAMMPS-GPU executions in G-HPCToolkit, we observed a 21% scalability loss attributed to `MPI_Allreduce` waiting for `cuInit`, using HPCToolkit's differential profiling capabilities. Repeated runs and micro benchmarks confirmed a systematic problem in `cuInit` on the same set of nodes. KIDS administrators reported that GPU power capping was enabled on some nodes. Subsequent system upgrades resolved the issue.

5.3 Stalls on Titan and KIDS

For our Titan case study, we again chose LAMMPS-GPU as our sample program. We ran LAMMPS-GPU on Titan using 1024 MPI ranks. Our blame-shifting idleness analysis showed no glaring algorithmic problems, but it weakly hinted at a problem with `MPI_Allreduce`. To investigate that problem, we looked at the lightweight traces. Figure 11 highlights a typical iteration (one of many) that suffered an unexpected delay. Visual inspection of the problematic `MPI_Allreduce` instances showed that one or two of the preceding CUDA synchronize operations took substantial time. Since a single rank behaving poorly can sabotage an MPI collective operation, we focused on the suspicious CUDA synchronize operations. At first, we suspected a hardware problem. This hypothesis, however, was easily ruled out as it was never the same MPI rank behaving badly. Further inspection of the trace data indicated that the offending stalls appeared to have very few samples. The absence of samples typically indicates a system call that blocked. Our stall analysis confirmed this. Furthermore, by inspecting the GPU traces, we saw that the GPU activity to which the CPU was synchronizing had already completed. This means that the extended sync waiting time was a complete waste. Our stall analysis also quantified the cumulative delay. While the delay was relatively minor for each instance of `MPI_Allreduce`,

System	Base running time in sec	Profiling time in sec (overhead%)		Tracing time in sec (overhead%)	
		HPCTOOLKIT	G-HPCTOOLKIT	HPCTOOLKIT	G-HPCTOOLKIT
KIDS	92.948	98.116 (5.56%)	98.038 (5.48%)	97.802 (5.22%)	98.036 (5.47%)
Titan	130.352	137.193 (5.25%)	138.062 (5.91%)	137.907 (5.80%)	138.368 (6.15%)

Table 3: Measurement overhead for LAMMPS-GPU on KIDS and Titan supercomputers.

the cumulative slowdown was about 17%.

The next question we considered was the scaling consequences of the phenomenon. The small study shown below indicates a scaling problem. There is a significant jump when moving from 256 nodes to 512.

Num MPI ranks	128	256	512	1024
Cumulative time wasted(%)	1.3	1.9	18.4	17.4

The next step in tracking down causes was to run LAMMPS-GPU on KIDS to see if we could observe the phenomenon on a different platform. On KIDS, for a 128-rank configuration, we observed the analogous stall again, but the cumulative time wasted showed more variation — between 10% and 30%.

Given that the blocking stalls occur on two separate platforms, we next questioned whether or not this phenomenon was peculiar to LAMMPS-GPU. To answer this question, we constructed a "proxy" app that just launched some GPU kernels, called CUDA synchronize, and then did a reduction. The proxy app showed the same random blocking stalls exhibited by LAMMPS-GPU.

Next we measured the proxy app *directly* on KIDS, using Intel x86 `rdtsc` instructions. For each iteration of the proxy test loop, we measured the time of CUDA synchronize operation on each of the 128 nodes. The results of this test showed that many iterations had one or two nodes with exceptionally long sync times. In addition, when comparing different iterations that had outliers, the MPI rank was not the same. This is exactly the same pattern as revealed by G-HPCTOOLKIT. Therefore, we concluded that our tool was not the source of the anomaly.

Given the nature of the blocking stalls, our intuition was that the synchronization strategy employed by the CUDA runtime/GPU driver combination might be suboptimal. Fortunately, CUDA has an API call (`cudaSetDeviceFlags`) for configuring the waiting strategy employed by a CUDA synchronization operation. A CPU can either yield-wait, spin-wait, or combined-yield-spin-wait for a GPU activity. The default CUDA runtime synchronization strategy is to spin-wait for a while, then yield-wait. On KIDS, we changed the default strategy to use exclusively spin-wait. Unfortunately, this change did not change the sporadic blocking phenomenon. Observing CUDA synchronization calls under `strace` revealed that the spin-waits still query the driver via `ioctl` system call.[‡] The system call can still block.

At this point, we need more data. All we know is that *something* systemic is degrading the performance of applications that use CUDA synchronization operations prior to `MPI_Allreduce`. The problem does not appear to be endemic to Titan. Further investigation will be needed to determine the underlying cause.

5.4 Preliminary Overhead Evaluation

Here, we present a very preliminary evaluation of the runtime overhead of G-HPCTOOLKIT on the KIDS and Titan supercomputers. The point of this preliminary evaluation was to confirm that our hybrid sampling-plus-instrumentation technique did not introduce unacceptable overhead. Our "at the terminal" intuition was favorable, but it was gratifying to see our initial impressions confirmed by a little data.

For our overhead study, we compare the overhead introduced by G-HPCTOOLKIT with that of the original unmonitored execution. We also compare G-HPCTOOLKIT's CPU+GPU monitoring overhead with that of HPCTOOLKIT's CPU-only monitoring to quantify the additional overhead introduced by our GPU performance measurement strategies. We used LAMMPS-GPU running on one CPU core utilizing one GPU for our empirical experiments. We measure our overhead for both profiling and tracing. We used the default provided `in.gpu.rhodo` LAMMPS input file, which is a rhodospin protein benchmark performing 200 time step simulation on 256000 atoms, for our experiments. Experiments were conducted with HPCTOOLKIT's default sampling rate of 200 samples per second. Table 3 shows the observed monitoring overhead. G-HPCTOOLKIT's CPU+GPU monitoring overhead is about 5% and it is in the ballpark of the original CPU-only monitoring overhead of HPCTOOLKIT for both profiling and tracing. The slightly higher overhead of G-HPCTOOLKIT is natural due to our wrapping of CUDA API calls and insertion of events into GPU streams to time kernels.

While these results are promising, they are not definitive. A detailed study of the monitoring overhead for a broad collection of applications is outside the scope of this paper.

6. RELATED WORK

Several strategies have been proposed [18, 21] that assess the kernel-level performance of GPUs. Less, however, has been done to provide holistic performance analysis of hybrid parallel applications on heterogenous supercomputers.

With regard to performance tools focused exclusively on GPU kernels, NVIDIA provides a state-of-the-art measurement-based tool — *Nvidia Visual Profiler (NVP)* [16]. NVP traces the execution of each GPU task, recording method name, start and end times, launch parameters, and GPU hardware counter values, among other information. NVP provides an extension called NVTX that allows programmers to manually instrument CPU-side events. Other tools, e.g., [3, 21], employ modeling and/or simulation to provide insight into the performance of individual kernels. Although these kernel-focused tools are *aware* of calls to CUDA host APIs on the CPU, none of these tools provide significant insight into CPU activity of heterogeneous applications.

With regard to performance tools focused on both GPU and CPU activities, two leading tools are TAU [11] and VampirTrace [7]. Both of these tools offer an array of techniques for performance analysis of hybrid architectures.

While TAU has some support for sampling, it principally

[‡] A collaborator at Nvidia has filed a driver bug report based on our observations [6].

employs instrumentation for measuring performance. Consequently, TAU has a natural integration path to absorb GPU instrumentation. On Nvidia devices, TAU uses CUPTI to both monitor execution of GPU tasks and capture GPU hardware counter values. TAU considers code performance from multiple perspectives, including inter-node communication, intra-node execution, CPU-GPU interactions, and GPU kernel executions.

The Vampir performance analysis toolset [7] traces program executions on heterogeneous clusters. VampirTrace monitors GPU tasks using CUPTI, logging information including kernel launch parameters, hardware counter values, and details about memory allocations. To monitor CPU activity, the Vampir toolset collects a trace of function entry/exit like TAU. Vampir toolset performance analysis of hybrid codes is based on post-mortem analysis of traces.

Our stall analysis resembles a capability in Scalasca [4]. Scalasca employs source instrumentation to trace communication events and performs trace replay to associate the cost of wait states to their causes. Our technique differs from Scalasca by being sampling based. Our technique identifies one specific case of waiting that is caused by blocking system calls. Our technique identifies the impact of late arrivers in only collective communications where as Scalasca is more generic and identifies causes of delays in both point-to-point and collective operations.

7. CONCLUSIONS AND FUTURE WORK

Based on our case studies, we draw the following conclusions:

- CPU-GPU blame shifting is an effective way to analyze algorithmic issues, especially those pertaining to division of work between CPU and GPU. Our case study of LULESH 1.0 [8] yielded a 30% improvement; LULESH 2.0 [9], developed with feedback from our tools, now avoids this problem.
- When performance issues are *not* work-partitioning related, lightweight sampling-driven traces are useful for diagnosing other problems, including hardware problems.
- There appears to be a systemic problem related to system calls blocking CUDA synchronization calls on both Titan and KIDS. This problem is detrimental to achieving scalable performance on these systems.
- We need better ways to extract performance data from the OS especially about blocking system calls. For example, the Solaris operating system provides an accounting of time spent inside a blocked system call when profiling with `ITIMER_REALPROF` [17]. Other approaches could involve tracing kernel calls or system-wide sampling, but these capabilities are not universally available.

Blaming kernels instantaneously has proven useful in our experience. However, future exascale applications might employ more asynchrony and need more sophisticated analysis to identify predecessors that delayed a GPU kernel charged using instantaneous blaming. Our blame-shifting technique identifies a "first-order" suspect. The true culprit, however, may be further removed. Consider the example in Figure 12.

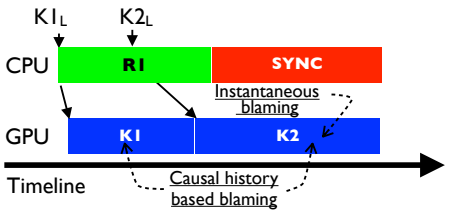

Figure 12: Blaming shifting vs. Full causal history.

In this case, blame shifting would identify kernel $K2$ as a candidate for tuning. Instantaneous blaming wouldn't attribute blame to kernel $K1$; however, reducing the time spent in $K1$ would also reduce CPU sync time. $K1$ can be blamed by examining the causal history, i.e., by keeping track of all tasks that executed on the GPU since the last point when the GPU went idle. As part of our future work, we intend to pursue higher-order causal analysis of idleness.

Other avenues of future work for us include:

- Extensively characterizing the overhead of our techniques, including both strong and weak scaling effects.
- Extending our techniques to OpenCL.
- Unifying our CPU-GPU blame shifting with the blame-shifting techniques for work-stealing and lock contention.
- Devising mechanisms and analysis approaches to gain more useful information about OS kernel activities, especially system calls that block.
- Supporting performance analysis in presence of GPU dynamic parallelism, e.g. where kernels launch other kernels [15].
- Working with vendors to correct the illusory concurrency problem noted in Section 4.5.

8. ACKNOWLEDGMENTS

We thank David Goodwin and Duncan Poole from Nvidia for their collaboration. This research was supported by the DOE Office of Science under Cooperative Agreement DE-FC02-07ER25800 and Sandia National Laboratory under Purchase Order 1293383. This research used resources of the National Center for Computational Sciences (NCCS) at Oak Ridge National Laboratory. NCCS is supported by the DOE Office of Science under Contract No. DE-AC05-00OR22725. This research used resources of the Keeneland Computing Facility at the Georgia Institute of Technology, which is supported by the National Science Foundation under Contract OCI-0910735.

9. REFERENCES

[1] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. HPCToolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, 22:685–701, April 2010.

[2] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *Proc. of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, PLDI '97, pages 85–96, 1997.

[3] S. S. Baghsorkhi, I. Gelado, M. Delahaye, and W.-m. W. Hwu. Efficient performance evaluation of memory hierarchy for highly multithreaded graphics processors. In *Proc. of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 23–34, 2012.

[4] D. Bohme, M. Geimer, F. Wolf, and L. Arnold. Identifying the root causes of wait states in large-scale parallel applications. In *In Proc. of the 2010 39th Intl. Conference on Parallel Processing*, ICPP '10, pages 90–100, Washington, DC, USA, 2010. IEEE Computer Society.

[5] W. M. Brown, P. Wang, S. J. Plimpton, and A. N. Tharrington. Implementing molecular dynamics on hybrid high performance computers - short range forces. *Computer Physics Communications*, 182(4):898–911, 2011.

[6] Dave Goodwin, NVIDIA Corp. Personal communication.

[7] D. Hackenberg, G. Juckeland, and H. Brunst. Performance analysis of multi-level parallelism: inter-node, intra-node and hardware accelerators. *Concurrency and Computation: Practice and Experience*, 24(1):62–72, 2012.

[8] I. Karlin et al. Lulesh programming model and performance ports overview. Technical Report LLNL-TR-608824, Lawrence Livermore National Laboratory, December 2012.

[9] I. Karlin, J. Keasler, and R. Neely. Lulesh 2.0 updates and changes. Technical Report LLNL-TR-641973, Lawrence Livermore National Laboratory, August 2013.

[10] C.-K. Luk, S. Hong, and H. Kim. Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Proc. of the 42nd Annual IEEE/ACM Intl. Symposium on Microarchitecture*, MICRO 42, pages 45–55, 2009.

[11] A. D. Malony et al. Parallel performance measurement of heterogeneous parallel systems with GPUs. In *Proc. of the 2011 Intl. Conference on Parallel Processing*, ICPP '11, pages 176–185, Washington, DC, USA, 2011. IEEE Computer Society.

[12] G. Nakhimovsky. Debugging and Performance Tuning with Library Interposers. `http://dsc.sun.com/solaris/articles/lib_interposers.html`, Jul 2001.

[13] NVIDIA Corp. CUDA Tools SDK CUPTI User's Guide DA-05679-001_v01, October 2011.

[14] NVIDIA Corp. NVIDIA CUDA C Programming Guide Version 4.1, October 2011.

[15] NVIDIA Corp. NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110, 2012.

[16] NVIDIA Corp. Nvidia Visual Profiler. `https://developer.nvidia.com/nvidia-visual-profiler`, Jan 2013.

[17] Open Solaris Forum. Man solaris - getitimer (2). `http://www.opensolarisforum.org/man/man2/getitimer.html`, June 2001.

[18] S. J. Pennycook, S. D. Hammond, S. A. Jarvis, and G. R. Mudalige. Performance analysis of a hybrid MPI/CUDA implementation of the NAS-LU benchmark. *SIGMETRICS Performance Evaluation Rev.*, 38(4):23–29, Mar. 2011.

[19] F. Petrini, D. J. Kerbyson, and S. Pakin. The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q. In *Proc. of the 2003 ACM/IEEE conference on Supercomputing*, SC '03, pages 55–, New York, NY, USA, 2003. ACM.

[20] S. Plimpton. Fast parallel algorithms for short-range molecular dynamics. *Journal of Computational Physics*, 117(1):1–19, Mar. 1995.

[21] J. Sim, A. Dasgupta, H. Kim, and R. Vuduc. A performance analysis framework for identifying potential benefits in GPGPU applications. In *Proc. of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 11–22, 2012.

[22] F. Song, S. Tomov, and J. Dongarra. Enabling and scaling matrix computations on heterogeneous multi-core and multi-GPU systems. In *Proc. of the Intl. Conference on Supercomputing*, ICS '12, pages 1–11, 2012.

[23] N. R. Tallent, J. Mellor-Crummey, M. Franco, R. Landrum, and L. Adhianto. Scalable fine-grained call path tracing. In *Proc. of the Intl. Conference on Supercomputing*, ICS '11, pages 63–74, 2011.

[24] N. R. Tallent, J. M. Mellor-Crummey, and M. W. Fagan. Binary analysis for measurement and attribution of program performance. In *Proc. of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 441–452, June 2009.

[25] N. R. Tallent, J. M. Mellor-Crummey, and A. Porterfield. Analyzing lock contention in multithreaded applications. In *Proc. of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 269–280, 2010.

[26] J. S. Vetter et al. Keeneland: Bringing heterogeneous GPU computing to the computational science community. *Computing in Science Engineering*, 13(5):90–95, Sept.–Oct. 2011.